Introduction to OpenMP

www.openmp.org
Previous Lectures

• Concurrent and distributed programming and it’s limitations
  • Amdahl’s vs. Gustafson's law

• Problem partitioning: SIMD vs. MIMD

• Memory architecture: UMA, NUMA and distributed
  • Memory-optimized programming
  • Memory latency vs. bandwidth

• Cache
  • Cache coherence
  • False sharing

• Granularity and load balancing
  • Task-management/synchronization overhead vs. strugglers.

• Concurrent algorithms
  • Communication cost of different implementation approaches
Motivation

• Parallelize the following code using threads:
  ```
  for (i=0; i<n; i++) {
    sum = sum + sqrt(sin(data[i]));
  }
  ```

• A lot of work to do a simple thing

• Different threading APIs:
  • Windows: CreateThread
  • UNIX: pthread_create

• Problems with the code:
  • Need mutex to protect the accesses to sum
  • Different code for serial and parallel version
  • No built-in tuning (# of processors someone?)
OpenMP

- A language extension that introduces parallelization constructs into the language
- Parallelization is orthogonal to the functionality
  - If the compiler does not recognize the OpenMP directives, the code remains functional (albeit single-threaded)
- Based on **shared-memory** multithreaded programming
- Includes constructs for parallel programming:
  - critical sections, atomic access, variable privatization, barriers, etc.
- Industry standard
  - Supported by Gnu, Intel, Microsoft, Sun, IBM, HP, and more compilers
  - Some behavior is implementation-dependent
OpenMP execution model

- Fork and Join: Master thread spawns a team of threads as needed
OpenMP basics

• Most constructs in OpenMP are compiler directives or pragmas.
• For C and C++, the pragmas take the form:
  
  ```c
  #pragma omp construct
  [clause [clause]…]
  ```

• Main construct:
  
  ```c
  #pragma omp parallel
  {
    ...
  }
  ```
  
  • Defines a parallel region over structured block of code
  • Threads block at end of region
OpenMP: Parallel Regions

double D[1000];
#pragma omp parallel
{
    int i; double sum = 0;
    for (i=0; i<1000; i++) sum += D[i];
    printf("Thread %d computes %f\n", omp_thread_num(), sum);
}

• Executes the same code several times (as many as there are threads)
  • What is the use of repeating the same work several times in parallel? Can use `omp_thread_num()` to distribute the work between threads.
  • How many threads do we have? `omp_set_num_threads(n)`. Or, environment variable `OMP_NUM_THREADS`

• D is shared between the threads, i and sum are private
OpenMP Compiler Switches

```bash
gcc -fopenmp prog.c -o prog

OMP_NUM_THREADS=2; prog
```
General Work Sharing Constructs

```c
answer1 = long_computation_1();
answer2 = long_computation_2();
if (answer1 != answer2) { ... }
```

• How to parallelize?
  • These are just two independent computations!

```c
#pragma omp parallel
#pragma omp sections
{
    #pragma omp section
    answer1 = long_computation_1();
    #pragma omp section
    answer2 = long_computation_2();
}
if (answer1 != answer2) { ... }
```
Work-sharing: the **for** loop

```c
#pragma omp parallel
#pragma omp for
for(i = 0; i < N; i++)
    c[i] = a[i] + b[i];
```

- Threads are assigned an independent set of iterations
- Threads must wait at the end of work-sharing construct
  - **nowait** can be used to prevent waiting, i.e.:
    ```c
    #pragma omp for nowait
    for( ; ; ) {...}
    ```
Comparison

• Sequential code

```c
for (int i=0; i<N; i++) { a[i]=b[i]+c[i]; }
```

• (Semi) manual parallelization

```c
#pragma omp parallel
{
    int id = omp_get_thread_num();
    int Nthr = omp_get_num_threads();
    int istart = id*N/Nthr, iend = (id+1)*N/Nthr;
    for (int i=istart; i<iend; i++) { a[i]=b[i]+c[i]; }
}
```

• Automatic parallelization of the for loop

```c
#pragma omp parallel
#pragma omp for
{
    for (int i=0; i<N; i++) { a[i]=b[i]+c[i]; }
}
```
Notes on `#parallel for`

- Only simple kinds of for loops are supported
  - One signed integer variable in the loop.
  - **Initialization**: `var = init`
  - **Comparison**: `var op last`
    - `op`: `<`, `>`, `<=`, `>=`
  - **Increment**: `var++`, `var--`, `var+=incr`, `var-=incr`
  - All of `init`, `last`, `incr` must be loop invariant

- Can combine the parallel and work sharing directives:
  `#pragma omp parallel for`
Optimizations of `#parallel for`

- Load balancing
  - If some iterations are faster, some processors may get idle
  - We don’t always know the distribution of work, may need to re-distribute *dynamically*
- Granularity
  - Thread creation and synchronization are costly
  - Assigning work to threads per-iteration is costly
- Trade-off between load balancing and granularity!

```c
#pragma omp parallel
{
    #pragma omp for
    for( ; ; ){
        [...]
    }
}
```
Assigning iterations

• The **schedule clause** affects how loop iterations are mapped onto threads
  • **schedule(static [,chunk])**
    • Blocks of iterations of size “chunk” to threads
    • Round robin distribution
    • By default, chunk=loop-count/#threads
    • **When to use:** Predictable and similar work per iteration
  • **schedule(dynamic [,chunk])**
    • Threads grab “chunk” iterations
    • When done with iterations, thread requests next set
    • By default, chunk=1
    • **When to use:** Unpredictable, highly variable work per iteration
  • **schedule(guided [,chunk])**
    • Dynamic schedule starting with large block
    • Size of the blocks shrink; no smaller than “chunk”
    • **When to use:** Special case of dynamic to reduce scheduling overhead
Example: What schedule to use?

• The function `TestForPrime` (usually) takes little time
• But can take long, if the number is a prime indeed
• What schedule to use? `static`, `dynamic` or `guided`?

```c
#pragma omp parallel for schedule ???
for( int i = 0; i <= end; i += 2 )
{
    if ( TestForPrime(i) ) gPrimesFound++;
}
```
Controlling Granularity

• `#pragma omp parallel if (expression)`
  • Can be used to disable parallelization in some cases (e.g. when the input is determined to be too small)

• `#pragma omp num_threads (expression)`
  • Control the number of threads used for this parallel region
OpenMP: Extents

- Each `#pragma omp parallel` creates a team of threads, which exist as long as the following block executes
  - `#pragma omp for` and `#pragma omp section` must be placed within a `#pragma omp parallel`

```c
#pragma omp parallel
{
    #pragma omp for
    { ... }
    #pragma omp sections
    { ... }
    foo();
}

foo() {
    #pragma omp bla_bla
}
```
OpenMP: Static Extent

• Static (lexical) extent of a construct (parallel/for/sections)
  • All the lines of code in the following block

```c
#pragma omp parallel
{
    #pragma omp for
    { ... }
    #pragma omp sections
    { ... }
    foo();
}
```

```c
foo() {
    #pragma omp bla_bla
}
```
OpenMP: Dynamic Extents

• Dynamic extent of a construct (parallel/for/sections)
  • All the locations reachable dynamically from a statement
  • The code of functions called from a parallelized region is in the region’s dynamic extent

```c
#pragma omp parallel
{
    #pragma omp for
    { ... }
    #pragma omp sections
    { ... }
    foo();
}
```

Dynamic extent

```c
foo() {
    #pragma omp bla bla
}
```
OpenMP Runtime System Optimization

- **Optimization**: If there are several `#pragma omp for` and/or `#pragma omp section` within the same parallel, the threads will not be destroyed and created again.

```c
#pragma omp parallel
{
    #pragma omp for
    for( ; ; ) { ... }
    #pragma omp for
    for( ; ; ) { ... }
}
```
OpenMP Runtime System Limitation

- **Problem:** A `#pragma omp for` is not permitted within a dynamic extent of a parallel section
  - Must include the inner `#pragma omp for` within its own `#pragma omp parallel`
  - Nested parallelism? (more on this in the tutorial)
  - The effect is implementation-dependent (will it create a new set of threads?)

```c
#pragma omp parallel
{
    #pragma omp for
    for ( ; ; ) {
        foo();
    }
}
```
OpenMP memory model

• Shared memory model
  • Threads communicate by accessing shared variables

• The sharing is defined syntactically
  • Any variable that is seen by two or more threads is shared
  • Any variable that is seen by one thread only is private

• Race conditions possible
  • Use synchronization to protect from conflicts
  • Change how data is stored to minimize the synchronization
OpenMP: Data Environment

• Shared Memory programming model
  • Most variables (including locals) are shared by default – unlike Pthreads!

    ```c
    int sum = 0;
    #pragma omp parallel for
    for (int i=0; i<N; i++) sum += i;
    ```
  • Global variables are shared

• Some variables can be private
  • Automatic variables inside the statement block
  • Automatic variables in the called functions
  • Variables can be explicitly declared as private
    • In that case, a local copy is created for each thread
Overriding storage attributes

- **private**
  - A copy of the variable is created for each thread
  - There is no connection between the original variable and the private copies
  - Can achieve the same using variables inside `{ }`

- **firstprivate**
  - Same, but the initial value of the variable is copied from the main copy

- **lastprivate**
  - Same, but the last value of the variable (at the last iteration) is copied to the main copy

```c
int i;
#pragma omp parallel for private(i)
for (i=0; i<N; i++) { ... }
```

```c
int x=666, i;
#pragma omp parallel
{
    #pragma omp for firstprivate(x) \ lastprivate(i)
    for (i=0; i<N; i++)
        if (data[i] == x) x++;
    if (data[i] == x) x++;
}
```
Reduction

```c
for (j=0; j<N; j++) {
    sum = sum + a[j]*b[j];
}
```

- How to parallelize this code?
  - `sum` is not private, but accessing it atomically is too expensive
  - Have a private copy of `sum` in each thread, then add them up
- Use the reduction clause!
  - Any associative operator must be used: +, -, | |, |, *, etc.
  - The private value is initialized automatically (to 0, 1, ~0 ...)
#pragma omp reduction

• Syntax:
  #pragma omp reduction (op:list)

• The variables in “list” must be shared in the enclosing parallel region

• Inside parallel or work-sharing construct:
  • A PRIVATE copy of each list variable is created and initialized depending on the “op”
  • These copies are updated locally by threads
  • At end of construct, local copies are combined through “op” into a single value and combined with the value in the original SHARED variable

```c
float dot_prod(float*a, float*b, int N) {
    float sum = 0.0;
    #pragma omp parallel for reduction(+:sum)
    for(int i=0; i<N; i++) {
        sum += a[i] * b[i];
    }
    return sum;
}
```
OpenMP Synchronization

\[ x = 0; \]

\[ \#pragma omp parallel \]

\[ x = x+1; \]

• What should the result be (assuming 2 threads)?
  • 2 is the expected answer
  • But can be 1 with unfortunate interleaving

• OpenMP assumes that the programmer knows what (s)he is doing
  • Regions of code that are marked to run in parallel are independent
  • If access collisions are possible, it is the programmer’s responsibility to insert protection
Synchronization Mechanisms

• Many of the existing mechanisms for shared programming
  • Single/Master execution
  • Critical sections, Atomic updates
  • Ordered
  • Barriers
  • Nowait (turn synchronization off!)
  • Flush (memory subsystem synchronization)
  • Reduction (already seen)
Single/Master

- **#pragma omp single**
  - Only one of the threads will execute the following block of code
  - The rest will wait for it to complete
  - Good for non-thread-safe regions of code (such as I/O)
  - Must be used in a parallel region

- **#pragma omp master**
  - The following block of code will be executed by the master thread
  - No synchronization involved
  - Applicable only to parallel sections

Example:

```c
#pragma omp parallel
{
  do_preprocessing();
  #pragma omp single
  read_input();
  #pragma omp master
  notify_input_consumed();

  do_processing();
}
```
Critical Sections

• `#pragma omp critical [name]`
  • Standard critical section functionality

• Critical sections are global in the program
  • Can be used to protect a single resource in different functions

• Critical sections are identified by the name
  • All the unnamed critical sections are mutually exclusive between themselves
  • All the critical sections having the same name are mutually exclusive between themselves
Shared variables

• Notify the compiler that the variable is shared

```c
float dot_prod(float* a, float* b, int N)
{
    float sum = 0.0;
    #pragma omp parallel for shared(sum)
    for(int i=0; i<N; i++) {
        sum += a[i] * b[i];
    }
    return sum;
}
```

• What’s the problem here?
Shared modifier cont’d

• Protect shared variables from data races
  
  ```c
  float dot_prod(float* a, float* b, int N)
  {
    float sum = 0.0;
    #pragma omp parallel for shared(sum)
    for(int i=0; i<N; i++) {
      #pragma omp critical
      sum += a[i] * b[i];
    }
    return sum;
  }
  ```

• Another option: use `#pragma omp atomic`
  
  • Can protect only a single assignment
  
  • Generally faster than critical
Atomic execution

• Critical sections on the cheap
  • Protects a single variable update
  • Can be much more efficient (a dedicated assembly instruction on some architectures)

• #pragma omp atomic
  update_statement

• Update statement is one of:
  • var = var op expr
  • var op= expr
  • var++, var--
  • The variable must be a scalar
  • The operation op is one of: +, -, *, /, ^, &, |, <<, >>
  • The evaluation of expr is not atomic!
Ordered

• `#pragma omp ordered` statement
  • Executes the statement in the sequential order of iterations

• Example:

```
#pragma omp parallel for schedule(static,1)
for (j=0; j<N; j++) {
    int result = heavy_computation(j);
    #pragma omp ordered
    printf("computation(%d) = %d\n", j, result);
}
```
Barrier synchronization

• `#pragma omp barrier`
  • Performs a barrier synchronization between all the threads in a team at the given point.

• Example:

  ```c
  #pragma omp parallel
  {
    int result = heavy_computation_part1();
    #pragma omp atomic
    sum += result;
    #pragma omp barrier
    heavy_computation_part2(sum);
  }
  ```
Controlling OpenMP behavior

- `omp_set_dynamic(int)/omp_get_dynamic()`
  - Allows the implementation to adjust the number of threads dynamically

- `omp_set_num_threads(int)/omp_get_num_threads()`
  - Control the number of threads used for parallelization (maximum in case of dynamic adjustment)
  - Must be called from sequential code
  - Also can be set by `OMP_NUM_THREADS` environment variable

- `omp_get_num_procs()`
  - How many processors are currently available?

- `omp_get_thread_num()`
  - A unique identifier of the thread

- `omp_set_nested(int)/omp_get_nested()`
  - Enable nested parallelism

- `omp_in_parallel()`
  - Am I currently running in parallel mode?

- `omp_get_wtime()`
  - A portable way to compute wall clock time
Performance Issues

• Idle threads do no useful work
• Divide work among threads as evenly as possible
  • Threads should finish parallel tasks at same time
• Synchronization may be necessary
  • Minimize time waiting for protected resources
• Parallelization Granularity may be too low
Load Imbalance

- Unequal work loads lead to idle threads and wasted time.
  - Need to distribute the work as evenly as possible!

```c
#pragma omp parallel
{
    #pragma omp for
    for(;;){
        [...]
    }
}
```
Synchronization

• Lost time waiting for locks
  • Prefer to use structures that are as lock-free as possible!
  • Use parallelization granularity which is as large as possible.

```c
#pragma omp parallel
{
    #pragma omp critical
    {
        [...] 
    }
    [...] 
}
```
Example: Parallel Numerical Integration

\[ f(x) = \frac{4.0}{1 + x^2} \]

```c
static long num_steps = 100000;
double step, pi;

void main()
{
    int i;
double x, sum = 0.0;

    step = 1.0/(double)num_steps;
    for (i=0; i<num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = \%f\n",pi);
}
```
Computing Pi through integration

• Parallelize the numerical integration code using OpenMP

• What variables can be shared?

• What variables need to be private?

• What variables should be set up for reductions?

```c
static long num_steps = 100000;
double step, pi;

void main()
{
    int i;
    double x, sum = 0.0;

    step = 1.0/(double)num_steps;
    for (i=0; i<num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n",pi);
}
```
Computing Pi through integration

```c
static long num_steps = 100000;
double step, pi;

void main()
{
    int i;
    double x, sum = 0.0;

    step = 1.0/(double)num_steps;
    #pragma omp parallel for private(x) reduction(+:sum)
    for (i=0; i<num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n",pi);
}
```

*i* is private since it is the loop variable
Conclusions

• Parallel computing is good today and indispensable tomorrow
  • Most today’s processors are multicore

• OpenMP: A framework for code parallelization
  • Available for C, C++ and FORTRAN
  • Based on a standard
  • Implementations from a wide selection of vendors

• Easy to use
  • Write (and debug!) code first, parallelize later
  • Parallelization can be incremental
  • Parallelization can be turned off at runtime or compile time
  • Code is still correct for a serial machine
Floyd’s Algorithm

• How do we parallelize it using OpenMP?

procedure FLOYD_ALL_PAIRS_SP(A):
1. \( D^0 = A \)
2. for \( k := 1 \) to \( n \) do
3. for \( i := 1 \) to \( n \) do
4. for \( j := 1 \) to \( n \) do
5. \( d^{(k)}_{i,j} = \min\{d^{k-1}_{i,j}, d^{k-1}_{i,k} + d^{k-1}_{j,j}\} \)

1. \textbf{void floyd}(A, n) \{ 
2. \hspace{1em} for (int k=0; k<n; k++)
3. \hspace{2em} for (int i=0; i<n; i++)
4. \hspace{3em} for (int j=0; j<n; j++)
5. \hspace{4em} A[i][j] = \min(A[i][j],
6. \hspace{5em} \downarrow A[i][k] + A[k][j]);
6. \}

Floyd’s Algorithm with OpenMP (1)

• Only one extra line
  • Can we do better?

1. `void floyd(A, n) {
2.   for (int k=0; k<n; k++)
2i.     #pragma omp parallel for
3.   for (int i=0; i<n; i++)
4.     for (int j=0; j<n; j++)
5.       A[i][j] = min(A[i][j],
4     A[i][k] + A[k][j]);
6. }
"
Floyd’s Algorithm with OpenMP (2)

• We create the parallel construct only once instead of every iteration
  • Can we do better?

1. `void floyd(A, n) {`
2i.  `#pragma omp parallel`
2.   `for (int k=0; k<n; k++)`
2i.  `#pragma omp parallel for`
3.   `for (int i=0; i<n; i++)`
4.   `for (int j=0; j<n; j++)`
5.   `A[i][j] = min(A[i][j],`
7.   `}`
Floyd’s Algorithm with OpenMP (3)

- We can order the runtime to parallelize two nested loops
  - We could have add another #pragma omp for in the nested loop
  - Can we do better?

1. `void floyd(A, n) {`
2i. `#pragma omp parallel`
2. `for (int k=0; k<n; k++)`
2i. `#pragma omp parallel for collapse(2)`
3. `for (int i=0; i<n; i++)`
4. `for (int j=0; j<n; j++)`
5. `A[i][j] = min(A[i][j], {
  A[i][k] + A[k][j]);`
6. `}`
Floyd’s Algorithm with OpenMP (4)

• What is the best schedule for this algorithm?

1. void floyd(A, n) {
2.     #pragma omp parallel
2i.     #pragma omp parallel for collapse(2)
3.         for (int i=0; i<n; i++)
4.             for (int j=0; j<n; j++)
5.                 A[i][j] = min(A[i][j],
4             A[i][k] + A[k][j]);
6. }